CS3210 Cheatsheet

github.com/reidenong/cheatsheets, AY25/26 Sem 1.

Parallel Computing Architectures

Parallelism can be present at different levels of architecture:

- Within a processor:
  - Bit level
  - Instruction level
  - Thread level
- Processor Level:
  - Shared memory
  - Distributed memory

Bit Level Parallelism

- Increasing processor word size to decrease number of instructions (64 bit `int` needs 2 operations in a 32 bit system but only 1 in 64 bit system)

Instruction Level Parallelism

Pipelining (Time parallelism)

- Instruction execution is split into multiple stages (eg. Fetch, Decode, Execute, Write). This allows multiple instructions to occupy different stages in the same clock cycle.
- Parallelism increases with number of pipeline stages, but faces limitations in data hazards and control hazards.

Superscalar (Space parallelism)

- The pipelines and hardware are duplicated, and the processor finds multiple independent instructions in an instruction sequence and executes them in parallel on execution units.
- Instructions come from the same execution flow (Thread). 2 Fetch/Decode units, 2 ALUs, 1 execution context.
- The limitation is in structural hazards arising from resource contention, eg. multiple instructions needing a single memory writing port.

SIMD (Single Instruction, Multiple Data)

- A type of parallelism where a single instruction operates on multiple data points simultaneously, given a vector architecture.
- Takes advantage of data-level parallelism by applying the same operation to multiple data elements at once.

Thread Level Parallelism

SMT (Simultaneous Multithreading)

- Refers to multiple logical cores running on a single physical core. Each logical core has its own execution state but shares the same physical resources.
- When there are pipeline bubbles from stalls/data hazards, other threads can utilize the resources.

Processor Level Parallelism

- Application with multiple execution flows can be mapped to multiple processors.

Flynn's Parallel Architecture Taxonomy

- SISD: Traditional uniprocessors
- SIMD: Vector instruction set (AVX), GPUs
- MISD: Multiple instruction streams working on the same data, pipeline like; rare
- MIMD: Each processing unit (core) fetches its own instruction and operates on its own data. Commonly used in multiprocessors

Some nVidia GPUs are noted to have SIMD + MIMD behaviour.

Multicore Processor Architecture

Hierarchical design

- Multiple cores share multiple caches, with cache size increasing from the leaves to the root. Commonly a private L1 cache (L1I, L1D) (~32 KB), private L2 cache (1 MB), shared LLC (32 MB).

Pipelined design

- A single data stream is processed by multiple execution cores in a pipelined way.
- Used when the same computation steps have to be applied to a long sequence of data elements.

Network-based design

- Cores, their local caches and memories are connected via an interconnection network.

Memory Organization

Terminology:

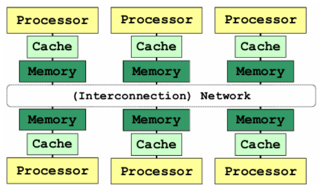
- Memory latency: Time for a memory request from a processor to be serviced by the memory system.
- Memory bandwidth: the rate at which the memory system can provide data to a processor.

Performant parallel programs should access memory infrequently.

- Organize computation to fetch data from memory less often, cache locality.
- Favor performing additional arithmetic to storing/loading values

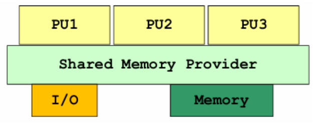
Distributed memory systems

- Each node is an independent unit with processor, memory.



Shared memory

- Programs/threads access memory through the shared memory provider, program is not aware of actual hardware memory architecture.



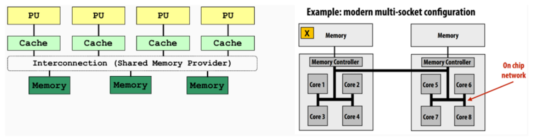
- Pros
  - No need to partition code or data
  - Communication is efficient as there is no need to move data among processors
- Cons
  - Synchronization is required
  - Lack of scalability due to contention

Uniform Memory Access Time (UMA)

- A multiprocessor organization where the latency of accessing the main memory is the same for every processor.
- Suitable for a small number of processing units due to contention (memory request collisions).

NUMA

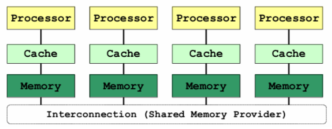
- Each PU has its own attached local memory, but with interconnection this local memory is available to other PUs as well in a global (distributed) shared-memory address space.
- Accessing local memory is faster than remote memory for a processor.



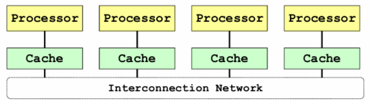
*UMA left, NUMA right

Other shared memory architecture

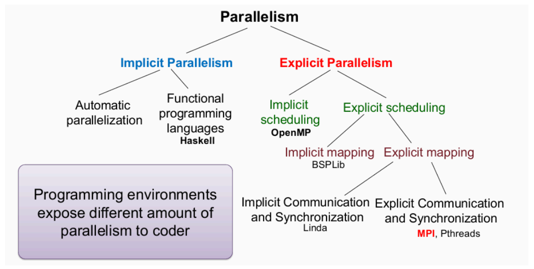
- ccNUMA: Cache coherent NUMA, each node has cache memory to reduce contention



- COMA: Cache only memory architecture, each memory block works as cache memory.



Parallel Programming Models


Parallelism
Implicit Parallelism
Explicit Parallelism
Automatic parallelization
Functional programming languages
Haskell
Implicit scheduling
OpenMP
Explicit scheduling
Implicit mapping
BSPLib
Explicit mapping
Programming environments expose different amount of parallelism to coder
Implicit Communication and Synchronization
Linda
Explicit Communication and Synchronization
MPI, Pthreads

Parallelism refers to the average number of units of work that can be performed in parallel per unit time. We define work as inclusive of the overhead of dependencies.

- Data parallelism: each processing unit carries out similar operations on its part of the data
- Task parallelism: partition the tasks in solving the problem among the processing units.

We may represent task dependences as a DAG, where a node represents each task and an edge represents a control dependency. Then the critical path length is the longest comletion time for any path, and the degree of concurrency = $\frac{\text{Total Work}}{\text{Critical Path length}}$.

Models of Coordination

Shared Address Space

- Tasks communicate by reading/writing to shared variables. This requires synchronization in the form of hardware support to implement efficiently. Bears a strong resemblance to **shared memory systems** (UMA, NUMA).
- very little structure

Data Parallel

We map a function to a large collection of data, the each operation is independent and as such no communication among the different functions is needed.

- Very rigid computation structure

Message Passing

- Tasks operate with their private address spaces, and communicate by explicitly sending and receiving messages. Bears a strong resemblance to **distributed memory systems**.
- Highly structured communication

Hardware Implementations

- It is common to implement message-passing abstractions on shared memory machines
  - Copying data to/from message library buffers
- It is possible to implement shared address spces abstractions on machines without hardware support
  - Maintaining a shared variable by message passing
  - Sharing a variable by having page-fault handler issue network requests

Program Parallelization

Fine grain vs Coarse grain parallelism

- Describes the frequency of coordination needed relative to the computation
- Fine grain: Relatively small work per task, little computation between coordination events
- Coarse grain: Large amounts of work per task, heavy computation between infrequent coordination events

Fosters Design Methodology

1. Partitioning

- Computation and data are divided for maximum parallelism
  - Data Centric (Domain decomposition) is data parallelism, divide data into pieces and associate computations with them
  - Computation Centric (Functional decomposition) is task parallelism, divide computation into tasks and determine how to associate data with computations
- At least 10x more primitive tasks than cores in target computer are required to minimize redundant computations and data storage
- Number of tasks an increasing function of problem size

2. Communication / Coordination

- Local communication: Task needs data from a small number of other tasks
- Global communication: Significant number of tasks contribute data to perform a computation

3. Agglomeration

- Combine tasks into larger tasks to eliminate communication between primitive tasks
- Increases locality of parallel algorithm

4. Mapping

- The assignment of tasks to execution units
- Tradeoff between
  - Maximizing processor utilization (increasing parallelism)
  - Minimizing inter-processor communication (placing tasks that communicate frequently on the same processing units to increase locality)
- Considerations could be between designs based on one task per core and multiple tasks per core
- If dynamic task allocator is chosen, the task allocator should not be a bottleneck to performance
- If static task allocator is chosen, the ratio of tasks to cores should be at least 10:1

Automatic Parallelization

Parallelizing compilers may perform decomposition and scheduling. However, there are several limitations:
- Difficult to analyze dependencies with pointer-based computations or indirect addressing
- Execution time of function calls or loops with unknown bounds are difficult to predict at compile time
- Complex memory hierarchies and opaque hardware behavior make the compilers less effective

Parallel Programming Patterns

Fork-Join
- Task creates child tasks which may run in parallel
- Implemented in processes, threads, etc.

Parbegin-parend
- Programmer specifies a sequence of statements to be executed by a set of cores in parallel
- When an executing thread reaches this construct, a set of threads are then created and the statements are assigned
- Like fork-join but all forks are at the same time and all joins are at the same time
- Implemented in language constructs such as OpenMP or compiler directives

SIMD
- Single instructions are executed synchronously by the different threads on different data (Vector registers, AVX Instructions)

SPMD
- Same program executed on different cores but operate on different data
- Different threads may execute different parts of the parallel program because of
  - Different speeds of the executing cores
  - Control statement in the program, eg. `if` statement
- No implicit synchronization, this is otherwise achieved by explicit synchronization operations
- Implemented in programs on GPGPU

Master-Worker
- A single program controls the execution of the program by assigning work to worker threads
- Good when worker threads are largely homogenous (same amount of work)
- Naive Implementation: Scatter tasks to all workers, compute, then gather all results/ new tasks
- Better Implementation: Task Pools
- Works well with star topologies

Task pools
- Number of threads are created statically by main thread. Work is not preallocated to worker threads, but retrieved from the task pool.
- Access to the task pool must be synchronized to avoid race-conditions (Producer-Consumer)
- Good when overhead for thread creation is independent of the problem size and the number of tasks, and when threads are heterogeneous
- Bad for fine grained tasks, as overhead of retrieval and insertion of tasks is non trivial

Pipelining
- Data in the application is partitioned into a stream of data elements that flows through the pipeline stages one after the other to perform different processing steps
- A form of functional parallelism (Stream parallelism)
- Best for task parallelism, and when each task takes roughly the same amount of time.

Producer-Consumer
- Producer threads produce data which are used as input by consumer threads, and they communicate through a shared data buffer

Performance of Parallel Systems

Performance goals
- Users: Reduced response time (end time - start time)
- Computer Managers: high throughput

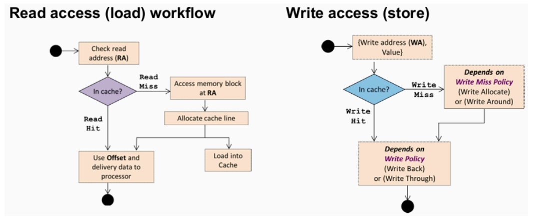
Performance of Sequential Programs
- Response time (wall-clock time) includes
  - User CPU time
  - System CPU time
  - Waiting time (I/O, time sharing)

For a program A,

$$\begin{aligned}&\text{Time}_{\text{user}}(A)\\&= \text{Time}_{\text{CPU cycle}} \times N_{\text{cycle}}(A)\\&= \text{Time}_{\text{CPU cycle}} \times N_{\text{instr}}(A) \times \text{CPI}(A)\end{aligned}$$

where CPI is the average number of CPU cycles needed for instructions in A, and depends on the internal organization of the CPU, memory system and compiler.

With Memory Access considerations


Read access (load) workflow
Check read address (RA)
In cache?
Read Miss
Access memory block at RA
Allocate cache line
Read Hit
Use Offset and delivery data to processor
Load into Cache
Write access (store)
{Write address (WA), Value}
In cache?
Write Miss
Depends on Write Miss Policy (Write Allocate) or (Write Around)
Write Hit
Depends on Write Policy (Write Back) or (Write Through)

Considering memory access to cache,

$$\begin{aligned}&\text{Time}_{\text{user}}(A)\\&= \text{Time}_{\text{CPU cycle}} \times \left(N_{\text{cycle}}(A) + N_{\text{mm cycle}}(A)\right)\end{aligned}$$

$$\begin{aligned}&N_{\text{mm cycle}}(A)\\&= N_{\text{read cycle}}(A) + N_{\text{write cycle}}(A)\\&= \underline{\left(N_{\text{read cycle}}(A) \times \text{Rate}_{\text{read miss}} \times (\text{Cycles to load cache line})\right)}\\&+ N_{\text{write\_cycle}}(A)\end{aligned}$$

Average Memory Access time

$$\begin{aligned}&\text{Time}_{\text{read access}}\\&= \text{Time}_{\text{read hit}} + (\text{Rate}_{\text{cache miss}}) \times \text{Time}_{\text{read miss penalty}}\end{aligned}$$

For a multi-level cache setup,

$$\begin{aligned}&\text{Time}_{\text{read access}}\\&= \text{Time}^{\text{L1}}_{\text{read hit}} + \text{Rate}^{\text{L1}}_{\text{cache miss}} \times \text{Time}^{\text{L1}}_{\text{read miss penalty}}\end{aligned}$$

$$\begin{aligned}&\text{Time}^{\text{L1}}_{\text{read miss penalty}}\\&= \text{Time}^{\text{L2}}_{\text{read hit}} + \text{Rate}^{\text{L2}}_{\text{cache miss}} \times \text{Time}^{\text{L2}}_{\text{read miss penalty}}\end{aligned}$$

We can get the global miss rate as

$$\text{Rate}_{\text{global read miss}} = \text{Rate}^{\text{L1}}_{\text{cache miss}} \times \text{Rate}^{\text{L2}}_{\text{cache miss}}$$

Throughput

Million-Instruction-Per-Second

$$\text{MIPS}(A) = \frac{N_{\text{instr}}(A)}{\text{Time}_{\text{user}}(A) \times 10^6} = \frac{\text{Clock Frequency}}{\text{CPI}(A) \times 10^6}$$

Since MIPS may not be meaningful, we may wish to consider MFLOPS if the goal is to maximize the throughput of FLOPs ona system.

$$\text{MFLOPS}(A) = \frac{N_{\text{fl ops}}(A)}{\text{Time}_{\text{user}}(A) \times 10^6}$$

Performance of Parallel Programs

Time

Let parallel execution time be given by $T_p(n)$, where T is the parallel execution time, there are p processing units, and the problem is of size n.

Then we have

$$\text{Speedup } S_{p(n)} = \frac{T_{\text{best sequential}}(n)}{T_p(n)}$$

In theory, $S_p(n) \leq p$, but in practice a superlinear speedup can occur if there are performance benefits from parallel executions (eg. overlapping of cache lines).

Cost

We can measure the total amount of work performed by all processors, ie. processor-runtime product.

$$\text{Cost } C_p(n) = p \times T_p(n)$$

A program is cost-optimal if it executes the same total number of operations as the fastest sequential program.

Efficiency

Efficieny is the actual degree of speedup performance achieved compared to the maximum.

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_{\text{best sequential}}(n)}{C_p(n)} = \frac{T_{\text{best sequential}}(n)}{p \times T_p(n)}$$

Scalability

Amdahl's law
- Speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelized, f (the sequential fraction)
- $0 \leq f \leq 1$.

From this, we have

$$S_p(n) = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Gustafson's law
- Sequential fraction f is not constant w.r.t problem size n, larger problem sizes may lead to smaller f and thus be more parallelizable.

This may be expressed by

$$\lim_{n \to 0} f(n) = 0 \Rightarrow \lim_{n \to 0} S_p(n) = p$$

Assumptions for laws
- Amdahl: f is fixed and known
- Gustafson: $f \to 0$ if the problem is large enough
- No parallelization overheads
- Identical processors
- Scaling does not increase overhead
- Memory not the main bottleneck

Scaling Constraints
- Problem size to machine ratio
  - For small problems, parallelism overhads may dominate parallelism benefits
  - For large problems, key working set may not 'fit' in small machine (thrashing to disk, key working set exceeds cache capacity, etc.)

Types of Scaling Problems
- Problem constrained scaling (PC): Solving the same problem faster, possibly using a parallel computer
- Time constrained scaling (TC): Completing more work in a fixed amount of time
- Memory constrained scaling (MC): running the largest problem posssible without overflowing main memory

Arithmetic Intensity

$$\text{Arithmetic Intensity} = \frac{\text{amount of computation (eg. instructions)}}{\text{amount of communication}}$$

A high arithmetic intensity is required to efficiently utilize modern parallel processors since the ratio of compute capability to available bandwidth is high.

Locality

Temporal locality
- The same memory location is accessed repeatedly within a short time window
- After the cache line containing the address is loaded, subsequent accesses hit the cache

Spatial locality
- Nearby memory locations are accessed close together in time.
- A cache line can hold multiple adjacent words, after accessing one element, the subsequent accesses to neighbours are hit

Taking advantage of locality

- Avoid sharing cache lines among tasks running on different cores in parallel
- Use padding to avoid cache line sharing
- Allocate work to tasks in a manner that takes advantage of prefetching

Performance Analysis

There are several ways to find possible performance bottlenecks

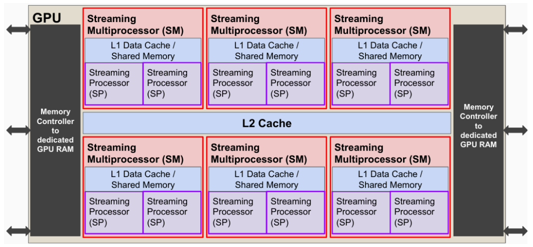
- **Instruction-rate limited (CPU bottleneck)**:
  - Test: add more computation (non-memory math instructions), if execution time grows linearly, the bottleneck is instruction throughput
- **Memory Bottleneck**:
  - Test: remove most computation but keep data loads, if execution time has little difference, memory is limiting performance
- **Data locality**:
  - Test: Force all array accesses to the same element (`A[0]`)
- **Synchronization overhead**:
  - Test: eliminate atomic operations and locks, see how much faster the code becomes.

GPGPU Programming

The CPU and GPU sit on the same motherboard and are connected via high speed links such as PCIe.

GPU Hardware Architecture

- GPUs have multiple **Streaming MultiProcessors (SMs)**, which are largely independent of each other. SMs have schedulers, register files, shared memory/L1 and multiple SPs (ALUs)
- Each SM consists of multiple compute cores (Streaming Processors), which are scalar ALUs
  - SPs execute instructions, and have access to the SM's register file and its memory.
- SMs share the same L2 cache and global memory, and atomics and memory fences ensure cooperative groups across SMs.


GPU
Streaming Multiprocessor (SM)
L1 Data Cache / Shared Memory
Streaming Processor (SP)
L2 Cache
Memory Controller to dedicated GPU RAM

CUDA Programming Model

- CUDA is massively hardware multithreaded, and is designed to scale well over time.
  - Very high arithmetic throughput via the SIMT model
- It is a general purpose programming model available as a simple extension to standard C, and has a mature software stack with both high-level and low-level access
- It launches batches of threads on the GPU.
- There is a high data-transfer and launch overhead as we need to move data between the host and device
  - For small tasks or tasks with heavy branching, CPU threading may outperform CUDA.

Compilation

- we use the `nvcc` compiler, which outputs both CPU code and GPU intermediate assembly code called PTX, but all in one binary
- When this binary is run, CPU code runs on the host CPU, while the kernel launches send work to the GPU, which executes the SASS instructions
- At runtime or during compilation, PTX gets compiled into SASS (Streaming Assembler) by a translator based on your specific GPU architecture.

Device vs Host

- Device refers to GPU
- Host refers to CPU
- Kernel refers to the initial functions that runs on the device
  - `__global__` prefix indicates a function is a kernel
  - `__device__` prefix is for other functiosn that runs on the device
  - `__host__` prefix is for CPU functions (this is default)

CUDA threads

- CUDA threads are logically organized into blocks, which are organized in grids. All threads run the kernel.
  - SIMT model: Single instruction, multiple thread
- CUDA threads are extremely lightweight
  - Very little creation overhead, instant switching, transparently scales to hundreds of cores and thousands of parallel threads
- Each block (containing multiple threads) is assigned to execute on one Streaming Multiprocessor for the entire kernel duration
  - These threads can have shared memory, atomic operations and synchronize within the block.

Logical Thread Hierarchy

- A kernel is executed by a grid of thread blocks.
- Each block has multiple threads, and cooperation only exists within each block.
- For scalability, the hardware is free to schedule thread blocks to any SM

CUDA Execution Model

Mappings

- Mappings:
  - Kernel $\rightarrow$ Runs on the whole GPU across multiple SMs
  - Thread block $\rightarrow$ Maps to a single SM
  - Thread instructions $\rightarrow$ Executed by SPs within the SM
  - Warp (32 threads) $\rightarrow$ Execution unit scheduled inside SM

SIMT

- Single-instruction, multiple thread
- SMs create, manage and schedule threads in SIMT warps. These threads in a warp start at the same program address Threads have individual program counter and register state. The warp executes one common instruction at a time
- Full efficiency is realized when all 32 threads of a warp agree on their execution path, ie. threads execute in lock-step. This achieves maximum instruction throughput
- Programmer's code may cause threads to diverge, lowering instruction throughput

CUDA Memory Model

- Each thread has up to 255 registers (Hopper). If per-thread data dosen't dit in registers, it uses local memory, which is much slower GPU DRAM.
- Each block has shared memory (Fast, L1 Cache)
- All blocks can access memory stored in slow but cached GPU DRAM
  - Global memory: read/write
  - Constant, Texture memory: readonly, better for linear accesses and 2D spatial accesses respectively.

GPU Memory Management

- Allocation: `cudaMalloc` , `cudaMemset` , `cudaFree`
- Copying Data: `cudaMemcpy`
- With CUDA's Unified Memory Model, we can use the prefix `__managed__` to automatically copy data between host and data when needed.

Synchronization

- `void __syncthreads();` acts as a barrier for all threads in a block

Example: Matrix Multiplication

- Each thread computes one output matrix value
- Each block computes some area of the result matrix

```
// MM code
int size = 400; // n = 400
int BLOCK_SIZE = 16;
dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE);
dim3 gridDim(
  (size + BLOCK_SIZE - 1) / BLOCK_SIZE,
  (size + BLOCK_SIZE - 1) / BLOCK_SIZE);
__global__ void matmul(
    float *A,
    float *B,
    float *C,
    int size) {
    int row =
      blockIdx.y * blockDim.y + threadIdx.y;
    int col =
      blockIdx.x * blockDim.x + threadIdx.x;
    if (row < size && col < size) { // No OOB
        float sum = 0.0f;
        for (int k = 0; k < size; k++) {
          sum +=
            A[row * size + k] *
            B[k * size + col];
        }
        C[row * size + col] = sum;
    }
}
```

- An improvement could be to use shared memory, as it is much faster than global memory

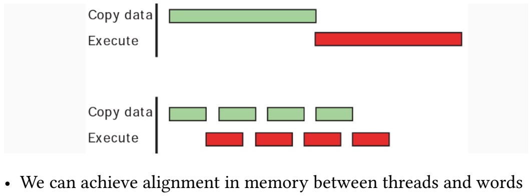
Optimizing CUDA Programs

Strategies

- Optimizing memory to achieve maximum memory bandwidth
- Maximizing parallel execution (data parallelism and hardware utilization)
- Optimizing instruction usage for maximum instruction throughput

Memory Optimizations

- Minimizing data transfer between host and device
  - Peak bandwidth betwen device memory and GPU is much higher than that between host memory and device memory
  - Should minimize data transfer between host and device memory
    - Prefer larger and less frequent transfers of memory from host to device
- Host memory can be paged out by the OS, forcing CUDA to make a extra staging copy before transferring. We can pin the memory to lock it in RAM in host, and this results in higher transfer speed with no extra copy
- We can also have overlapping asynchronous transfers with computation with `cudaMemcpyAsync()` . This uses different streams to achieve concurrent copy and execute for greater parallelism



- We can achieve alignment in memory between threads and words by coalescing simultaneous accesses to global memory by threads in a warp. This reduces the number of data transactions needed.

Maximising Parallel executions

- Number of warps should be larger than number of multiprocessors, so all SPs have at least one warp to execute
- Threads per block should be multiple of warp size, minimum of 64 threads
  - Avoids memory bank conflicts, facilitates coalescing
- Ensure that at least one block can run on an SM
  - When a thread block allocates more registers than available on a multiprocessor, the kernel launch fails
- Avoid multiple contexts per GPU
  - context-switching is very slow because the drivers must save memory mappings, kernel state and flush caches

Maximize Instruction Throughput

- Minimize the use of arithmetic instructions with low throughput, potentially trading precision for speed
  - Single-precision floats provide the best performance
  - Integer division and modulo operations are particularly costly, replace with bitwise operations when possible
  - Use signed loop counters so C can apply more aggressive optimizations
- Minimize divergent warps caused by control flow
- Optimize out synchronization points to reduce the number of instructions overall

Memory Consistency

In the shared address space model, tasks communicate by reading and writing to shared variables. This requires hardware support to scale and can be rather costly.

Cache Coherence

Multiple copies of the same data exist on different caches, and local data updates by the processor should be visible to other processors. Coherence ensures that each processing unit has a consistent view of memory through its local cache.

Properties of a coherent memory system:

1. Program order
   - A single core should see its memory read/writes in program order
2. Write propagation
   - Writes from a core should become visible to other cores eventually
3. Transaction serialization
   - All writes to a location are seen in the same order by all processing units

Maintaining Memory Coherence

Cache coherence can be maintained by:

1. Software based solutions
   - OS + Compiler + hardware aided solutions
     - eg. OS uses page fault to propagate writes
2. Hardware based solutions

Cache Properties

- As cache size increases, access time increases due to addressing complexity, but cache misses are Reduced
- Data is transferred between main memory and cache in sizes of the cache line size. Larger blocks reduces the number of blocks, replacements last shorter, but also increase the chance of spatial locality hits.
- Typical L1 cache line size is 4-8 words

Write policies

- Write through: write access is immediately transferred to main memory
  - +: we always get the newest value of a memory block
  - -: High number of memory accesses make this very slow, though a write buffer may make things better
- Write back: Write operations are performed in only in the cache, and we only write to main memory when we are replacing the cache block
  - +: Less write operations
  - -: Memory may contain invalid entries

Tracking cache line Sharing Status

- Snooping based: No centralized directory, each cache keeps track of the sharing status by monitoring the bus
  - All the processing units on the bus can observe every bus transaction (Write Propagation)
  - Bus transactions are visible to the processing units in the same order (Write serialization)
  - Granularity of the cache coherence is at the block level
- Directory based: Sharing status is kept in a centralized location. Commonly seen on NUMA architectures

Cache Coherence drawbacks

- Overhead in shared address space, CC appears as increased memory latency in multicore processor
- Multiple processing units may read and modify the same global variable, resulting in cache ping pong (back and forth updating)

False Sharing

- When multiple threads write to logically disjoint elements on the same cache line, the cache line bounces back and forth between the caches of the two processors
- Results in a large number of invalidations from writes to shared lines, resulting in many misses

Memory Consistency

- Constraints the order in which memory operations performed by one thread become visible to other threads for different memory locations, for example how writes and reads propagate among processing units.
- Consistency is present at both compile time (compilers) and run time (hardware)
- The consistency model is important to allow programmers to reason about correctness and program behaviour

Types of Consistency

Sequential Consistency

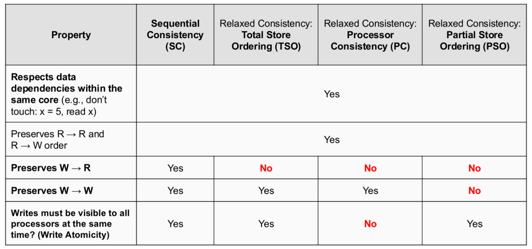
- The global result of all memory accesses of all processing units appear to all nodes in the same sequential order

Relaxed Consistency

- Relaxes the ordering of memory operations if there are no data dependencies
- Increases performance but also complexity

Weak Ordering

- No completion order of the memory operations is guaranteed



Property	Sequential Consistency (SC)	Relaxed Consistency: Total Store Ordering (TSO)	Relaxed Consistency: Processor Consistency (PC)	Relaxed Consistency: Partial Store Ordering (PSO)
Respects data dependencies within the same core (e.g., don't touch: x = 5, read x)	Yes			
Preserves R → R and R → W order	Yes			
Preserves W → R	Yes	No	No	No
Preserves W → W	Yes	Yes	Yes	No
Writes must be visible to all processors at the same time? (Write Atomicity)	Yes	Yes	No	Yes

Distributed Programming Models

Memory Consistency

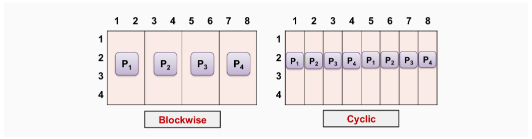
- Distributed systems do not face conventional problems of cache coherence and hardware memory consistency, as each processor has its own private memory with no hardware shared address space.
  - Cache updates are done through explicit messages
- Distributed systems may still need to manage consistency outside of memory and cache, eg. in file system and in general managing software level data consistency.

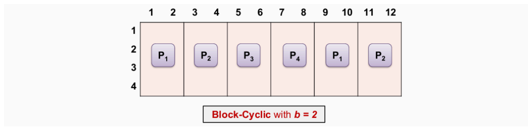
Data Distribution

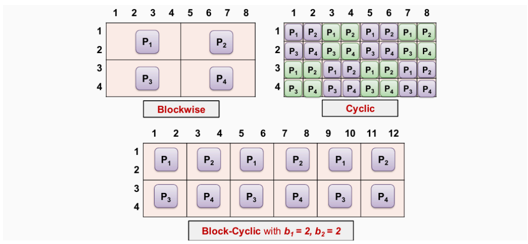
- Blockwise: Each node takes contiguous sections of data, good for programs that operate on spatially adjacent elements
- Cyclic: Each node is given data in a round-robin method, good when computation per value is expensive and we require good load balancing

2D Data Distribution

- With 2D data distribution we are also concerned with the needs for communication between nodes, ie. size of the data border.







Information Exchange

- Distributed Memory programming have disjoint memory spaces, and require data exchange through communication operations, which are point-to-point or global communication

Message Passing Model

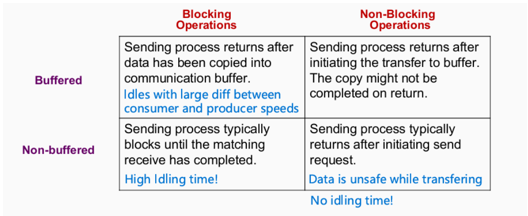
- Data is explicitly partitioned, and all interactions require both parties to participate
- Loosely synchronous
  - Synchronous: tasks may synchronize when communicating with each other
  - Asynchronous: other than this tasks execute completely synchronously

Communication Protocols

Definitions

- **Blocking**: a blocking call only returns when the resources used in the call may be re-used safely by the programmer
- **Buffered**: a buffered call copies user-data into a internal buffer and then returns control to the user, trading off space for time
- **Synchronous**: a synchronous send can complete only when a matching receive has started to execute; Requires coordination with other processes; 'Local' behaviour
- **Asynchronous**: a asynchronous send can complete regardless of communication with other processes; 'Non-local behavior'

Types of communication



	Blocking Operations	Non-Blocking Operations
Buffered	Sending process returns after data has been copied into communication buffer. Idles with large diff between consumer and producer speeds	Sending process returns after initiating the transfer to buffer. The copy might not be completed on return.
Non-buffered	Sending process typically blocks until the matching receive has completed. High Idling time!	Sending process typically returns after initiating send request. Data is unsafe while transfering

No idling time!

- Relevant Overheads
  - Non-buffered: Idling
  - Buffered: Buffer management
- Tradeoffs
  - Blocking: Program is easier to reason about
  - Non-blocking: Hidden communication overhead
- Coordination
  - Local/global view: Synchronous/Asynchronous

Message Passing

Message Passing Interface (MPI)

- A specification for message passing libraries
- Deadlocks can occur in MPI when message passing cannot be completed
  - Possibly when runtime system does not use system buffers or system buffers too small and both sends cannot complete
- An MPI program is called secure if the correctness of the program does not depend on assumptions about specific properties of the MPI runtime system
  - Eg. Deadlock-free logical rings, even nodes perform send → receive, odd nodes perform receive → send.

Process Groups and Communicators

- A process group refers to an ordered set of processes, where each has a unique rank.
- A communicator is a handle that processes can use to communicate with each other
  - Intra-communicators support the execution of arbitrary point to point communication operations on a single group
  - Inter-communicators support the communication operations between two process groups
- Groups and communication provide logical separation of processes based on tasks, such as enabling collective communication operations across a subset of related processes

Collective Communication

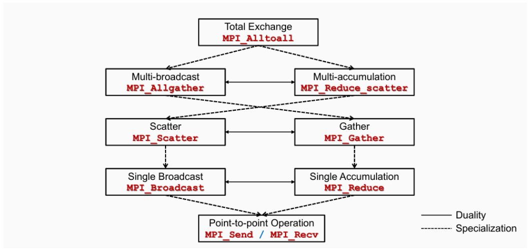
- Operations that involve all proceses in a communicator. This prevents deadlock from occurring if all processors participate correctly
  - MPI runtime internally manages message matching and synchronization for collectives

Duality of Communication Operations

- Consider a spanning tree where nodes are processes and directed edges are where communication occurs
- Two communication operations are a duality if the same spanning tree can be used for both operations

Stepwise Specialization

- Communication operations can be ordered into a hierarchy from the most general to the most specific
  - Specific operations can be implemented in terms of the more general ones



Interconnection Networks

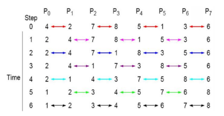
Interconnects form the backbone of communication between

- Intra-process components (cores, caches, memories)
- Processors and other processors (multi-socket systems)
- Nodes to other nodes (physically separate systems)

Parallel Sorting Algorithms

Linear

- Odd even transposition sorts can sort a linear array in $O(n)$ rounds with n processors



2D

- Shear sort
  - Sort by row (odd ascending, even descending)
  - Sort by column (all ascending)
  - Repeat
  - $\log N + 1$ phases, overall $\sqrt{N} \cdot (\log N + 1)$ steps (row/column sorts are done with transposition sorts)

Interconnect Topologies

- Direct Interconnection (Static): Endpoints are of the same type (cores/memory)
- Indirect Interconnection (Dynamic): Interconnect is formed by switches, not links

Direct Interconnects

Topology Metrics

- Diameter
- Degree (Max degree for graphs)
- Bisection Width: Minimum number of edgees to be removed to dibide the network into two equal halves
  - Acts as a measure of the network's capacity in the worst case (The smaller the bisection width, the easier the network becomes a bottleneck under heavy load)
  - High-performance interconnects aim for large bisection width
- Node connectivity: Minimum number of nodes that must be removed to disconnect the network
- Edge connectivity: Minimum number of edges that must fail to disconnect the networks

Special Topologies

- Hypercubes have good performance in general
  - Low diameter of $\log P$ for P processes
  - High bisection bandwidth; Many disjoint paths exist, so many pairs can communicate simultaneously without contention.
  - Low degree of $\log P$ for P processes
  - Easy to embed binary trees: at stage x pair nodes differing in stage x
- Complete graphs are not viable in practice
  - $\Theta(P^2)$ links needed
  - Routing hardware is too expensive with high node degrees

Cube-Connected-Cycles (CCC)

- From a k-dimensional hypercube, we can substitude each node with a cycle of k nodes to form a CCC with $k \cdot 2^k$ nodes.
- Each node is labeled as (X, Y), where X is the node index in the hypercube and Y is the position in the cycle
- Then node (X, Y) is connected to
  - $(X, (Y + 1) \bmod K)$
  - $(X, (Y - 1) \bmod K)$
  - $(X \oplus 2^Y, Y)$

Summary

network G with n nodes	degree $g(G)$	diameter $\delta(G)$	edge-connectivity $\mathrm{ec}(G)$	bisection bandwidth $B(G)$
complete graph	$n-1$	1	$n-1$	$\left(\frac{n}{2}\right)^2$
linear array	2	$n-1$	1	1
ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d-dimensional mesh ($n = r^d$)	$2d$	$d(\sqrt[d]{n} - 1)$	d	$n^{\frac{d-1}{d}}$
d-dimensional torus ($n = r^d$)	$2d$	$d \left\lfloor \frac{\sqrt[d]{n}}{2} \right\rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-dimensional hyper-cube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k-dimensional CCC-network ($n = k2^k$ for $k \geq 3$)	3	$2k - 1 + \lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
complete binary tree ($n = 2^k - 1$)	3	$2 \log \frac{n+1}{2}$	1	1
k-ary d-cube ($n = k^d$)	$2d$	$d \lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

Indirect Interconnects

- Reduced hardware costs by sharing switches and links
- Able to connnect two different types of hardware (eg. processors, memory) as switches provide indirect connections and can be configured dynamically

Metrics

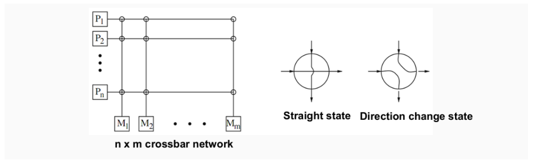
- Cost: Number of switches/links
- Concurrent connections

Bus

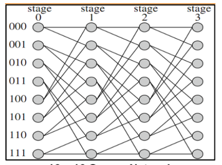
- A set of wires to transport data from sender to receiver
- Multiple devices but only one pair can communicate at a time
- Bus arbiter is used for coordination, typically used for a small number of processors

Multistage Switching Network

- Several intermediate switches with connecting wires between neighbouring stages
- Obtain a small distance for arbitrary pairs of input and output devices
- **Crossbar Network**
  - Costly hardware $(n \cdot m)$ switches for a small number of processors (16^2 for 16 processors and 16 memory nodes)
  - Used on Intel supercomputer processors

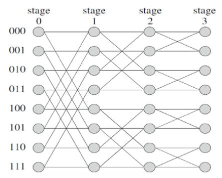


- **Omega Network**
  - One unique path for every input to output
  - A $n \times n$ Omega network has $\log n$ stages, each with $\frac{n}{2}$ of 2×2 switches (2 inputs, 2 outputs)
  - For each switch at (α, i) there is an edge to $(\alpha \ll, i + 1), (\alpha$ left shift with LSB inversion$, i + 1)$
  - $\frac{n}{2}$ switches per stage, $\log n$ stages, 32 switches for 16 processors and 16 memory nodes

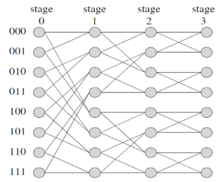


16 × 16 Omega Network

- **Butterfly Network**
  - (α, i) connects to $(\alpha, i + 1), (\alpha \oplus i\text{-th bit from left}, i + 1)$



- **Baseline Network**
  - (α, i) connects to (cyclic right shift of last (k-i) bits of $\alpha, i +$ 1) and (inversion of the LSB of $\alpha +$ cyclic shift of last (k- i) bits$, i + 1)$



Routing Algorithms

Terminology

- Minimal: Shortest path is always chosen
- Deterministic: Always uses the same path for the same (source, destination) pair
- Adaptive: Takes into account network state and adapts accordingly (congestion avoidance, avoid dead nodes)

Routing Examples

- 2D Mesh: just keep moving in the direction to minimize manhattan distance
- Hypercube: Fixing one bit at a time, at most N hops
- Omega Network: XOR-tag Routing
  - Let $T =$ source $\oplus$ destination
  - At stage k, if $(T \gg k)\&1$ crossover, else go straight.

Energy Efficient Computing

- Higher power levels usually give us more performance, but incur tradeoffs in costs and heat
- Dennard Scaling: The (false) theory that processors could always fit more transistors per unit area without using more power per unit area
- Processors cannot fit more transistors into the same space without using more power, but they cannot use more power due to thermal constraints

Per Process Efficiency

- Performance-per-Watt: Using a *score* of workload using another metric (GFLOPS, benchmarks), we do $\frac{\text{total score}}{\text{CPU power}}$.
- Performance (through increasing clock frequency) often does not increase linearly with power, faces diminishing returns
- Processor voltage has a more significant effect on processor power and thus temperature

$$\text{Total Power} = P_{\text{dynamic}} + P_{\text{static}}$$
$$P_{\text{dynamic}} = k \times V^2 \times f$$

- The total power of a processor consists of power used by the processor for operations and the power used indepedent of the processor
- The power for operations is a function of Processor voltage V, processor frequency f along with coefficient k, which is a value depending on the complexity of the program being run and the processor hardware.
  - Voltage has a significant impact on power
  - Voltage and frequency are not independent; higher frequencies require higher voltages for the processor to operate normally

Dynamic Voltage and Frequency Scaling

- Modern processors can adjust their voltage and clock frequency dynamically to reduce power usage and heat

Heterogeneous Cores

- Putting two types of processors together, for high performance and low performance
  - High performance chips need more power at lower performance levels
  - We can run latency-critical tasks on high-performance cores and low priority tasks on energy efficient cores
  - Intel: P cores (more efficient at higher performance levels), E cores (more efficient at lower performance levels)
  - Apple: 10 Complex performance cores, 4 simpler efficiency cores
- CPU/GPU are also considered heterogenous, and GPUs perform much better than CPUs at certain tasks, for example massively parallel workloads of SIMT computations

Datacenter Efficiency

Metrics

- GFLOPS-per-watt: Similar to per-processor efficiency, but across large multi-node benchmarks
- Power Usage Effectiveness: Overall data center energy efficiency (Energy used for computer vs. total energy usage)

$$\text{PUE} = \frac{\text{Total facility energy}}{\text{IT Equipment energy}} = 1 + \frac{\text{Non IT facility energy}}{\text{IT Equipment energy}}$$

Trends

- High Bandwidth CPU-GPU connections, possibly 5x less power used for data transfer with Grace Hopper chips

Improving Power Usage Effectiveness

- Aisle containment for heat management: Using hot aisle/cold aisle for efficient cooling

Appendix

Processes, Threads, and Synchronization

Program Parallelization

1. Decomposition of sequential algorithm
2. Scheduling tasks to Processes/Threads
3. Mapping processes to physical processors

Inter-process Communication

- Shared memory (Need to protect against race conditions)
- Message passing
  - Blocking vs non-blocking
  - Synchronous vs Asynchronous

Process Interaction with OS

- Exceptions can be cause by machine level instructions
  - Synchronous
- Interrupts can be caused by external events
  - Asynchronous (independent of program execution, have to execute an interrupt handler)

Threads

- A process may have multiple independent control flows called threads.
- A thread defines a sequential execution stream within a process, and have their own program counter, stack, and registers.
- All threads belonging to the same process see the same value and have shared memory architecture.
- Global variables of a program and all dynamically allocated data objects can be accessed by any thread of this process (Share text, data, heap)

Synchronization

Data race

1. Two concurrent threads access a shared resource without any protection
2. At least one thread modifies the shared resource.

Semaphores

- Mutual exclusion through atomic counters

```python
# Implementation of Counting Semaphore with Binary semaphore
class Csem:
    def init(k: int):
        val = k
        mutex = 1
        gate = 0

    def lock(Csem):
        wait(gate)
        wait(mutex)
        val = val - 1
        if val:
            signal(gate)
        signal(mutex)

    def signal(Csem) -> None:
        wait(mutex)
        if val == 0:
            signal(gate)
        val += 1
        signal(mutex)
```

Deadlock

Conditions for Deadlock

1. Mutual exclusion: at least one resource must be held in a non sharable mode
2. Hold & wait: There must be oneprocess holding one resource and waiting for another resource
3. No pre-emption: Critical sections cannot be aborted externally
4. Cyclical wait

Classical Synchronization Problems

Producer Consumer

```python
mutex = S(1)
dq = deque()
items = S(0)  # Finite buffer
spaces = S(k)


def consumer():
    wait(items)
    wait(mutex)
    consume(dq.popleft())
    signal(mutex)
    signal(spaces)


def producer():
    x = produce()
    wait(spaces)
    wait(mutex)
    dq.append(x)
    signal(mutex) # Signal mutex first to prevent context switching inefficiency
    signal(items)
```

Readers Writers

```python
readers = 0
mutex = S(1)
room_empty = S(1)

def reader():
    wait(mutex)
    if not readers:
        wait(room_empty)
    readers += 1
    signal(mutex)

    read()

    wait(mutex)
    readers -= 1
    if not readers:
        signal(room_empty)
    signal(mutex)

def writer():
    wait(room_empty)
    write()
    signal(room_empty)
```

Readers Writers with Lightswitch and Turnstile

```python
class Lightswitch:
    def __init__(self) -> None:
        self.counter = 0
        self.mutex = Semaphore(1)

    def wait(self, sem: Semaphore) -> None:
        wait(self.mutex)
        self.counter += 1
        if self.counter == 1:
            wait(sem)
        signal(self.mutex)

    def signal(self, sem: Semaphore) -> None:
        wait(self.mutex)
        self.counter -= 1
        if self.counter == 0:
            signal(sem)
        signal(self.mutex)

room_empty = Semaphore(1)
turnstile = Semaphore(1)

def writer():
    wait(turnstile)
    wait(room_empty)
    write()
    signal(room_empty)
    signal(turnstile)

def reader():
    wait(turnstile)
    signal(turnstile)
    read_lightswitch.lock(room_empty)
    read()
    read_lightswitch.signal(room_empty)
```

Implementing Locks

- locks can be implemented with both test-and-set and compare-and-swap.

```python
# Test and Set
# tas(x): return the previous value of x and set it to 1 atomically
def acquire(lock):
    while tas(lock):
        pass

def release(lock):
    lock = 0

# Compare and Swap
# cas(curr, old, new): if curr == old, set to new and return True.
def acquire(lock):
    while not cas(lock, 0, 1):
        pass

def release(lock):
    lock = 0
```

`perf`

Flag	Effect
`-r`	Repeat test, take the average
`-e`	Events of interest: • `LLC-loads` • `LLC-load-misses` • `LLC-dcache-load-misses` • `cycles` • `instructions` • `task-clock` • `branches` `load` is replaceable by `store`

* Low IPC could signal high synchronization/atomic contention

OpenMP

- OpenMP uses a thread pool model: threads are created once and used across parallel regions; Low thread startup overhead
- Shared memory programming is simple, as all threads see the same address space, and you can parallelise loops with minimal code changes

Parallelizing `for` loops

```c
// Parallelizing outer loop
// (a, b, result) are shared for threads
// (i, j, k) are private per-thread
void mm(matrix a, matrix b, matrix result) {
  int i, j, k;

  #pragma omp parallel
    for shared(a, b, result)
    private (i, j, k)
  for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
      for (k = 0; k < size; k++)
        result[i][j] += a[i][k] * b[k][j];
}
```

Arbitrary number of worker threads

We can write code that forks # of worker threads based on the number of cores available.

```c
#pragma omp parallel
{
  int thread_id = omp_get_thread_num();

  if (thread_id == 0) { // master
    std::cout << omp_get_num_threads();
  }
}
```

Work sharing constructs

There are several ways we can natively partition work among different threads

Loop iterations

- n iterations of the for loop are divided into pieces of size chunksize and assigned to the threads.
- **static**: work are assigned to the threads before running.
  - lower overhead of assignment as it is done statically at runtime
  - good when amount of work per task is likely to be equal
- **dynamic**: work is assigned to each thread when dynamically when it becomes free.
  - good when amount of work per task has large variance, so dynamic assignment allows for faster runtimes

```c
int chunk_size = 2;

#pragma omp parallel
{
  #pragma omp for schedule (static,
                            chunk_size)
  for (int i = 0; i < n; i++) {
    x[i] = y[i];
  }
}
```

Sections

- We manually define code blocks that can be assigned to available thread

```c
#pragma omp parallel sections {
    #pragma omp sections {
      work1();
    }
    #pragma omp sections {
      work2();
    }
  }
}
```

Synchronization Constructs

- `#pragma omp barrier` : synchronizes all threads to wait at barrier
- `#pragma omp master` : specifies a region executed only by master thread
- `#pragma omp critical` : specifies a block that must be executed only by one thread at most
- `#pragma omp atomic` : critical section but for a single assignment expression
- `omp_lock_t` : lock used to protect critical sections

Network File system

- A Network File System lets files be stored on one machine but accessed over a network as if they were local. Applications use normal file I/O without needing to know the files are remote.
- These are good for centralized storage, sharing, and access control.

Implementations

- **Client-server Model**
  - The storage server exports a directory, and the client mounts it into its local filesystem namespace
  - In mounting, a existing filesystem is attached to a directory path as a reference, not a copy. File operations to there are trapped by the Virtual File System (VFS) layer, which manages network RPC calls to the NFS server, which excecutes the operations locally and returns results over the network.
  - OS kernel translates local system calls into network requests
- **Protocol**
  - File operations are turned into RPCs, which run over TCP/IP
- **Consistency**
  - Clients may cache file data and metadata for performance
  - Servers enforce consistency using callbacks or periodic revalidation
- **Concurrency**
  - File locking APIs are extended over the network

CUDA Programming

Choosing Dimensions

1D Problem of size N:

```
// HOST CODE
dim3 block(16, 16); // 256 threads per block
dim3 grid( (N + block.x - 1) / block.x );
kernel_1D<<<grid, block>>>(d_A, d_B,d_C, N);

// KERNEL CODE
__global__ void kernel_1D(
    float *A, float *B, float *C, int N) {

    int idx =
    blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < N)
        do_work(idx);
}
```

2D Problem of size $Rr \times Cc$

```
// HOST CODE
dim3 block(16, 16);
dim3 grid(
    (Cc + block.x - 1) / block.x,
    (Rr  + block.y - 1) / block.y
);
// columns, rows
kernel_2D<<<grid, block>>>(d_A, d_B, d_C, Rr, Cc);

__global__ void kernel_2D(float *A, float *B,
float *C, int Rr, int Cc)
{
    int row =
    blockIdx.y * blockDim.y + threadIdx.y;
    int col =
    blockIdx.x * blockDim.x + threadIdx.x;

    if (row < Rr && col < Cc) {
        int idx = row * Cc + col;
        do_work(row, col);
    }
}
```

MPI Programming

Overview

6 basic functions

- `MPI_Init` : Starts the MPI env
- `MPI_Comm_size` : Returns number of processes
- `MPI_Comm_rank` : Returns rank of current process
- `MPI_Send` : Sends a message
- `MPI_Receive` : Receives a message
- `MPI_Finalize` : Shuts down the MPI env

Blocking	Non-blocking

`MPI_Send`
`MPI_Recv`
`MPI_Sendrecv`
`MPI_Sendrecv_replace`
	`MPI_Isend`
	`MPI_Irecv`

- Blocking and non blocking can be mixed
  - `MPI_Recv` can receive from `MPI_Isend`
  - `MPI_Irecv` can receive from `MPI_Send`
- `MPI_Sendrecv` is an MPI call that sends a message and receives a message in a single, atomic operation.
  - Designed specifically to avoid deadlock in two way communication patterns

Blocking Send and Receive

```
#include <stdio.h>
#include <mpi.h>
#include <string.h>

int main(int argc, char **argv)
{
    int rank, size, tag, i;
    char message[20];

    // Initialize environment, get own rank
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    tag = 100;

    if (rank == 0) {
        // Master node
        strcpy(message, "Hello World 2");
        for (i = 1; i < size; i++)
            MPI_Send(message, 14, MPI_CHAR,
              i, tag, MPI_COMM_WORLD);
    } else {
        // Slave node
        MPI_Status status;
        MPI_Recv(message, 14, MPI_CHAR,
          0, tag, MPI_COMM_WORLD, &status);
    }

    printf("node %d : %.13s\n",
           rank, message);

    MPI_Bcast(&data, 1, MPI_INT, 0,
MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

Non Blocking Send and Receive

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rank, p, size = 8;
    int left, right;
    char send_buffer1[8], recv_buffer1[8];
    char send_buffer2[8], recv_buffer2[8];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    /* Fill buffers with hostnames*/
    gethostname(send_buffer1, size);
    gethostname(send_buffer2, size);

    left  = (rank - 1 + p) % p;
    right = (rank + 1) % p;

    MPI_Request reqs[4];
    MPI_Status  stats[4];

    MPI_ISend(send_buffer1, size, MPI_CHAR, left,
TAG_LEFT, MPI_COMM_WORLD, &reqs[0]);
    MPI_IRecv(recv_buffer1, size, MPI_CHAR, right,
TAG_LEFT, MPI_COMM_WORLD, &reqs[1]);

    MPI_ISend(send_buffer2, size, MPI_CHAR, right,
TAG_RIGHT, MPI_COMM_WORLD, &reqs[2]);
    MPI_IRecv(recv_buffer2, size, MPI_CHAR, left,
TAG_RIGHT, MPI_COMM_WORLD, &reqs[3]);

    /* Wait for all 4 operations to complete */
    MPI_Waitall(4, reqs, stats);

    printf("Rank %d: left=%d, right=%d, recv1=%s,
recv2=%s\n",
             rank, left, right, recv_buffer1,
recv_buffer2);

    MPI_Finalize();
    return 0;
}
```

- `MPI_Waitall` takes in an array of `MPI_Request` objects and blocks until every request is complete.

MPI Programming Paradigms

SPMD

- All ranks run the same program but operate on different partitions of the data

Master-Worker

- Master maintains task queue and worker queue
- Good for dynamic load distribution, irregular computations

Data Parallel (Weather simulation)

- Compute on local partition, communicate results, synchronize and then perform next iteration
- Good for simulations, iterative solvers

Pipeline/Stream Parallelism

- Each rank performs a stage of a pipeline
- Good for multi-stage transformations, streaming computations

Collective Based Communication Patterns

- Used when the computation repeatedly depends on collectives

Pseudocode Examples

SPMD

```
MPI_Init()
rank := MPI_Comm_rank()
P    := MPI_Comm_size()

// Partition data
chunk := partition(data, P, rank)

// Compute locally
local_result := f(chunk)

// Combine
global_result := MPI_Reduce(local_result, op=SUM,
root=0)

if rank = 0 then
   print global_result

MPI_Finalize()
```

Master-Worker

```
MPI_Init()
rank := MPI_Comm_rank()
P    := MPI_Comm_size()

worker_queue := [1, 2, ..., P-1]
results := []

if (worker) {
  while true:
    t := MPI_Recv(from = 0)
    if t = TERMINATE: break
    r := solve(t)
    MPI_Send(r, to = 0)
}

if (master) {
  while tasks_remaining:
    if worker_queue is not empty:
        worker := pop(worker_queue)
        t := next_task()
        MPI_Send(t, to = worker)
    else:
        (r, worker) := MPI_Recv(from = ANY_WORKER)
        store result r
        push(worker_queue, worker)

// After all tasks assigned: wait for all
outstanding results
while worker_queue.size < (P-1):
    (r, worker) := MPI_Recv(from = ANY_WORKER)
    store result r
    push(worker_queue, worker)

// Send termination message
for w in 1..P-1:
    MPI_Send(TERMINATE, to = w)
}

MPI_Finalize()
```

Data Parallel

```
MPI_Init()
rank := MPI_Comm_rank()
state := init_local_state(rank)
for iter = 1 to MAX_ITERS:
  compute_local(state)
  halo := extract_boundary(state)
  MPI_Sendrecv(halo, left, recv_from = right)
  MPI_Sendrecv(halo, right, recv_from = left)
  MPI_Barrier()
MPI_Finalize()
```

Pipeline/Stream

```
MPI_Init()
rank := MPI_Comm_rank()
P    := MPI_Comm_size()

if rank = 0:
    for item in input_stream:
        MPI_Send(item, to=1)
else if rank = P-1:
    while true:
        x := MPI_Recv(from=rank-1)
        if x = END: break
        y := final_stage(x)
        output(y)
else:
    while true:
        x := MPI_Recv(from = rank-1)
        if x = END:
            MPI_Send(END, to = rank+1)
            break
        y := stage_process(rank, x)
        MPI_Send(y, to = rank+1)

MPI_Finalize()
```

Collective Based Communication Patterns

```
MPI_Init()
rank := MPI_Comm_rank()
P    := MPI_Comm_size()

if rank = 0:
    A := full_array
else:
    A := ∅

// Each rank gets a block of equal size
block_size := n / P
local_data := MPI_Scatter(A, block_size, root=0)

// Now compute statistics
local_sum := sum(local_data)
local_n   := size(local_data)

global_sum := MPI_Allreduce(local_sum, op=SUM)
global_n   := MPI_Allreduce(local_n, op=SUM)

mean := global_sum / global_n

MPI_Finalize()
```

Synchronization

- The only collective synchronization operation
- Processes block until all processes of the communicator have started the MPI_Barrier call

```
int MPI_Barrier(MPI_Comm comm);
```

Timing

```
double x = MPI_Wtime();
// Get relative wallclock time
```

Communication Operations

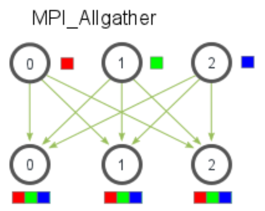
Broadcast

- Root sends the same data to all processes

```
int MPI_Bcast(void* buf, int count, MPI_Datatype
dt, int root, MPI_Comm c);
```

Multi-Broadcast

- Each processor sends the same data block to every other processor



```
int MPI_Allgather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int
recvcount, MPI_Datatype recvtype, MPI_Comm comm);
```

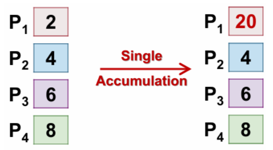
Scatter/Gather

- Partition / Acccumulates the data from/to a root node.

```
int MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int
recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm);
int MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int
recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

Reduction

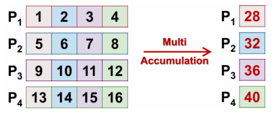
- Reduces data to a single root node



```
int MPI_Reduce(void *sendbuf, void *recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm);
```

Multi Accumulation

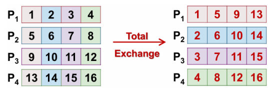
- Each processor scatters data to other blocks, then reduces those.



```
int MPI_Reduce_scatter(void *sendbuf, void
*recvbuf, const int recvcounts[], MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm);
```

Total Exchange

- Each processor does scatter.



```
int MPI_Alltoall(void *sendbuf, int sendcount,
MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype
recvtype, MPI_Comm comm);
```